Software de segmento terreno de próxima generación

Pablo Soligo, Jorge Salvador Ierache

Universidad Nacional de La Matanza
Departamento de Ingeniería e Investigaciones Tecnológicas.
Florencio Varela 1903, B1754JEC San Justo, Buenos Aires.
psoligo@unlam.edu.ar
http://www.unlam.edu.ar/

Resumen Este trabajo presenta los resultados de las experiencias obtenidas en el desarrollo de la primera versión de un prototipo funcional de segmento terreno de próxima generación. El proyecto fue inicialmente desarrollado como parte de la materia "proyecto integrador" de la Maestría en Desarrollos Informáticos de Aplicación Espacial (MDIAE) y evolucionado por el grupo de investigación aplicada de la Universidad Nacional de La Matanza (UNLAM). Las unidades de software desarrolladas permiten la ingestión y decodificación de telemetría de múltiples misiones con independencia del origen de datos y el comando y control de satélites. Se aplicaron técnicas y herramientas como camino alternativo a las tecnologías propietarias para desarrollar un entorno multimisión y multiplataforma. Como segmento de vuelo se ha utilizado el satélite de formación 2017 (FS2017) [1] y datos de misiones de terceros (Satnogs).

Keywords: Satelites Artificiales, Segmento Terreno, Diseño de Software, Tecnologia Espacial, Sistemas Distribuidos, Telemetría, Telecomandos

1. Introducción

El presente trabajo detalla las experiencias y resultados en el desarrollo de un prototipo de sistema de segmento terreno satelital costo-efectivo. El diseño e implementación del prototipo tiene como raíz las experiencias realizadas durante las maestrías con sistemas de segmentos terrenos actualmente en producción y su vinculación con los sistemas de software de propósito general de uso extendido en la industria.
Se priorizo el desarrollo sobre herramientas de amplia aceptación y de probada robustez en contraste con soluciones comunes en la industria espacial, las cuales en algunos casos muestra un exceso de conservadurismo [2]. Parte del objetivo de este trabajo es confrontar dichas implementaciones, con predominancia en tres dominios:

- **Lenguajes de decodificación, comando y control:** Se propone el uso de lenguajes e intérpretes de propósito general para la decodificación de telemetría y el desarrollo de scripts de comandos.

- **Persistencia** [1] **de datos:** Se plantea una arquitectura donde la persistencia de datos se realiza sobre motores de base de datos relacionales y su acceso es siempre mediante un ORM(Object Relational Mapper).
- **Interfaces:** Las interfaces con sistemas internos y externos, tanto para la comunicación de módulos como para publicación e ingestión de datos se realiza mediante protocolos extendidos y aceptados.

Se marginan de la solución, estándares, formatos o protocolos *ad-hoc* o de escasa penetración en la industria de software de uso masivo y se prioriza las soluciones, bibliotecas, componentes y frameworks COTS(Commercial Off-The-Shelf). Los lineamientos de diseño adhieren a los manuales de buenas practicas del área espacial [3], casi todas ellas comunes y aplicables a las buenas practicas conocidas en la industria de software general.

2. Lenguajes de decodificación, comando y control

Una practica común, en el área espacial, cuando se trata de comandar un satélite, es el uso de secuencias de comandos sobre estructuras de control. Empresas y agencias espaciales en todo el mundo han desarrollados sus propios lenguajes e intérpretes para cumplir con este objetivo [4]:

- STOL: Satellite Test and Operation Language. Desarrollado por la Nasa y ampliamente utilizado en varias misiones.
- PLUTO: presente en algunas misiones de la ESA (Satellite Control and Operation System 2000).
- Otros: desarrollados o utilizados por diferentes compañias SOL(GMV), CCL (Harris), PIL (Astrium), SCL (ICS).

Estos lenguajes son típicamente propietarios e incompatibles entre sí [5], dificultando las migraciones entre sistemas y la posibilidad de compartir procedimientos entre distintas misiones.
El FS2017 [1] fue el primer segmento de vuelo utilizado como prueba experimental para el desarrollo de los sistemas planteados en este trabajo y se optó por usar un lenguaje de propósito general en lugar de crear un lenguaje especifico o utilizar los existentes en CONAE (Comisión Nacional de Actividades Espaciales).

Este enfoque presenta múltiples ventajas, el cuadro 1 compara los pros y contras de cada tipo de implementación.

En el caso del proyecto integrador la opción de lenguaje propio de la misión se descartó por no disponer del tiempo que implica desarrollar y validar un intérprete de propósito específico. La opción de utilizar los intérpretes de CONAE quedó relegada por sobre la opción Python, con una base de usuarios mucho más grande. El índice de popularidad TIOBE [6] ubica al lenguaje en la posición 5. Incluso en su versión más popular, las alternativas de propósito especifico, al ser parte de una única o un grupo de organizaciones siquiera figuran

[1] Se refiere a la propiedad de los datos para que estos sobrevivan de alguna manera.

Cuadro 1. Comparativo Lenguajes Propósito especifico vs Propósito General

Propósito específico.	**Propósito general.**
Ventajas	**Desventajas.**
Pueden ser más amigables para usuarios no programadores. Pueden presentar adaptaciones especificas a los problemas de operación del satélite. Pueden presentar adaptaciones especificas a los problemas de operación del satélite.	Es una tecnología propietaria. Problemas de portabilidad. Requiere desarrollo propio. Requiere mantenimiento y las mejoras o ampliaciones son costosas. Muy baja base de usuarios.
Desventajas	**Ventajas.**
Demasiado poderoso. Menos amigable a la lectura.	Portable. Mas poderoso que las versiones de propósito específico. Gran cantidad de usuarios.

entre las primeras 50.
Utilizando Python también se consigue generar scripts multiplataforma, mejores capacidades de depuración, documentación disponible en la comunidad y un lenguaje de dominio de todos los estudiantes.
El uso de los interprete disponibles en CONAE sufre de las mismas restricciones que cualquier lenguaje de uso interno y de propósito especifico, una base de usuarios restringida a un grupo acotado y finito, documentación limitada, capacidades circunscritas del lenguaje y las herramientas, pero por sobre todo, y quizá lo más evidente, es la dificultad o el esfuerzo que implica de adaptar el intérprete a las nuevas plataformas como indica [4].

3. Persistencia de datos

Las definiciones de los datos (Alarmas, variables de telemetría, comandos disponibles, etc.) como los datos en sí, en cualquier nivel de procesamiento, están administrados por un motor de base de datos relacional. Para la telemetría satelital existen entidades para el almacenamiento de los datos crudos, independientemente del satélite al que pertenezcan, la fuente o el origen de la ingestión. A medida que estos datos son procesados pasan a tablas donde están las variables de ingeniería ya procesadas y disponibles para cualquier modulo que las necesite, incluyendo el procesador de comandos.
Previendo para futuras versiones la implantación de tablas que permitan el acceso eficiente a datos históricos, KPIs (Key Performance Indicators) y desnormalizaciones que permitan el almacenamiento de largo plazo.
Todo el acceso a datos se realiza exclusivamente a través de un ORM, el uso de esta capa intermedia ofrece además de una mejora en la productividad, una independencia del proveedor del motor de base de datos.
La solución propuesta en términos de persistencia, prácticamente un estándar de facto en la industria del software, aun no tiene presencia en muchos centros

de control de misión. Su incorporación, exploración y dominio representan un valor agregado al trabajo.

4. Interfaces

En términos de interfaces tanto internas como externas en el área espacial abundan las soluciones *ad-hoc* o implementaciones sobre tecnologías de escasa penetración.
Un caso remarcable es el de CORBA [7] (Common Object Request Broker Architecture), que, si bien es un estándar propuesto por varios jugadores de la industria del software, no ha sido adoptado masivamente y se vio rápidamente relegado por implementaciones basadas en HTTP (Hypertext Transfer Protocol) [8].
Las interfaces del sistema propuesto se basan completamente en esta última tecnología, por ejemplo, toda ingestión de telemetría se realiza mediante un servicio REST que puede ser consumido tanto por módulos de terceros como por módulos internos que adapten formatos, tecnologías o características de antiguas implementaciones.
La figura 1 muestra como los adaptadores ajustan dos tecnologías diferentes (TCP/IP y FTP) a una entrada normalizada sobre servicios REST. Así mismo la evolución del sistema a una arquitectura distribuida prevé distribución, sincronización y comunicación de tareas mediante brokers basados también en http.

5. Arquitectura

La solución propuesta para el segmento terreno evoluciona desde su concepto cliente-servidor tradicional a una arquitectura de sistema distribuido multiprocesador. Un subconjunto finito de procesos se ejecuta de manera distribuida dentro de la infraestructura de hardware disponible. La tabla 2 enumera algunos de los servicios a ejecutar.

Esta arquitectura permite escalar horizontalmente en función de las misiones que se puedan ir sumando al centro de control. La figura 1 muestra un diagrama de alto nivel donde se pueden observar adaptadores que ajustan la telemetría provista por distintas tecnologías a un estándar REST (Representational state transfer). Los procesos 1, 2 ... n son los especificados en la tabla 2.

En términos de desarrollo el lenguaje utilizado es Python, normalizando en un único lenguaje el desarrollo de los módulos de software, los scripts de comandos y las funciones de calibración.
El framework utilizado es Django, lo que motiva un análisis y diseño orientado a objetos con un alta productividad tanto desde el framework como en el lenguaje [10].
A esto ultimo se debe agregar la capacidad de incorporar a los scripts de comandos y de calibración el acceso a herramientas de comunicación como Sockets,

Cuadro 2. Servicios distribuidos

Proceso	**Descripción**
Central.	Servicio central, responsable de ofrecer las interfaces, tanto sea de usuario como las API de comunicación.
Procesador de Telemetría (n).	Proceso responsables de la decodificación de la telemetría cruda.
Procesador de Telecomandos (n)	Procesos responsables de la ejecución de comandos
Servicios CODS	Responsable de generar y actualizar las efemérides, eclipses y oportunidades de contacto (Pasadas).
Procesos de datos	Procesamiento de datos, desnormalizaciones, consolidados, estadísticas y tendencias[9].
Procesos de calibración	Verificar novedades en funciones de calibración.
Otros	Procesos adicionales a ser relevados.

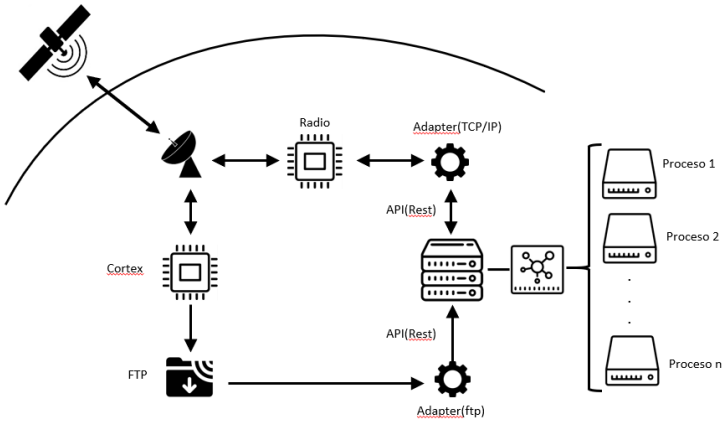

Figura 1. Arquitectura conceptual

http, RPC (Remote Procedure Call), Web Services, email y a bibliotecas de manejo de archivos XML entre otros como explica [4].

6. Telemetría

Si bien la arquitectura del segmento terreno es agnóstica de la fuente de datos se han utilizado como segmento de vuelo el FS2017 y datos de la misión LITUANICASAT 2. El FS2017 envía la telemetría en tramas AX.25 que son recibidas por un equipo de radio y publicadas por TCP/IP. La trama tiene un fragmento de Payload donde viene codificada la telemetría. La telemetría como su definición (Tipo, límites, rangos, ubicación) están almacenadas en tablas de un motor de base de datos relacional. Es común en este tipo de sistemas recibir los valores crudos o *raw*, para poder transformar esta telemetría cruda a valores de ingeniería es necesario someterlos a una función que realice la conversión, la cual también podría cambiar en función del desgaste o recalibración del sensor.

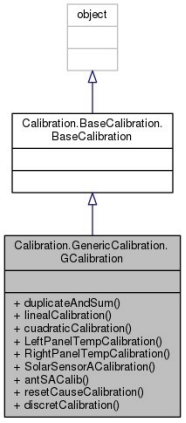

Figura 2. Calibración

Estos ajustes pueden ir desde un simple ajuste por función lineal hasta una discretización de valores por tablas de *look-up*. El sistema debe ser lo suficientemente flexible como para permitir aplicar cualquier función de transformación a toda variable de telemetría y permitir ajustarla en el tiempo.
En lugar de utilizar interpretes de desarrollo propio, la solución propuesta, utiliza técnicas de reflexión de software para cargar en tiempo de ejecución la función seleccionada de calibración y aplicarla al valor *raw* extraído de la trama de telemetría.
La función de calibración puede ser cualquier secuencia de comandos programable en python sin ningún tipo de restricción más allá del tiempo de procesamiento. Todas las bibliotecas y estructuras de datos están disponibles incluyendo funciones matemáticas e incluso el acceso a la base de datos completa mediante ORM. Al tener acceso a la base de datos se pueden obtener coeficientes actualizados de calibración permitiendo:

- **Reutilizar funciones:** Es posible crear una única función para ajustes típicos (lineal, cuadrático) y reutilizarla con distintos coeficientes según la variable de telemetría requiera
- **Calibración fina:** Creada una función es posible modificar los coeficientes para a medida que el sensor presenta desgaste.

Para que un método sea considerado de calibración o ajuste debe estar desarrollado como método de una clase heredada de *BaseCalibration*. El software

realiza periódicamente una exploración de todos los métodos públicos de clases derivadas (Ver procesos tabla 2) de *BaseCalibration* y los disponibiliza para su aplicación a las distintas variables de telemetría. La figura 2 muestra la clase *GCalibration*, heredada de *BaseCalibration*, donde se implementan algunos métodos de calibración incluido el mencionado *linealCalibration*.
El siguiente fragmento de código muestra parte de la implementación del método, encargado de llamar a la función de calibración para todo nuevo valor *raw*.
Calibración de variable raw

```
class TlmyVar(models.Model):
  def setValue( self,
raw,
saveifchange=False,
dt=datetime.now(utc)):

    value = self.tlmyVarType.getValue()
    if  (raw!=self.tlmyVarType.lastRawValue) or
        (self.tlmyVarType.lastUpdate==None):
      self.tlmyVarType.lastRawValue = raw
if self.tlmyVarType.calibrationMethod:
  if not self.tlmyVarType.calibrationLogic:
    klass =
      globals()[self.tlmyVarType.calibrationMethod.aClass]

    instance = klass()

    methodToCall =
      getattr(instance,
      self.tlmyVarType.calibrationMethod.aMethod)
    self.tlmyVarType.calibrationLogic = methodToCall
  else:
    pass #Calibracion ya cargada

          ...
```

(Extracción parcial del método setValue)

Si el valor *raw* para una variable determinada no cambió desde la última actualización y tampoco se recargo la función, entonces se guarda el registro de la recepción, pero no se aplican las calibraciones. Esto último hace ganar eficiencia ya que no se vuelve a transformar cuando se sabe que el resultado será el mismo. La carga del método de calibración a un atributo del tipo de variable de telemetría se realiza, si no ocurren cambios, una única vez durante la ejecución. Esto también permite mejorar los tiempos de ejecución ya que evita la carga por cada valor de telemetría recibido. Con una frecuencia configurable, el software analiza si alguna función de calibración fue actualizada, si así ocurriera se procede al

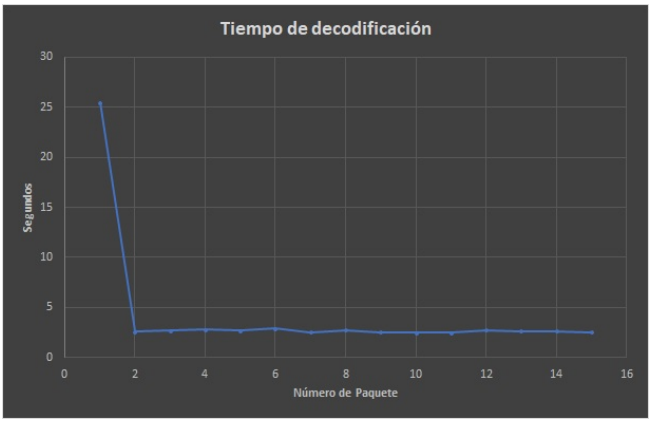

Figura 3. Tiempo de decodificación

limpiar el atributo que contiene la función para forzar su recarga.
La figura 3 muestra los tiempos de procesamiento para un conjunto de simulación de 15 paquetes con 5000 variables de telemetría cada uno, cantidad compatible con un satélite científico de envergadura (Ejemplo SABIAMAR `http://www.conae.gov.ar/index.php/espanol/introduccion-sace`).
La primera decodificación demora más que las siguientes dado que tiene que realizar la carga de las funciones de transformación. La primera telemetría recibida difiere de la anterior, ya que al ser la primera calibración la función no está precargada. Luego el tiempo se estabiliza entre 3 y 3.5 segundos para el conjunto de las 5000 variables con una probabilidad de cambio de 10 % entre muestra y muestra. El tiempo de procesamiento incluye además la verificación de que los valores de las variables de ingeniería estén dentro del rango de seguridad y el almacenamiento en el histórico. Las pruebas han sido realizadas en máquinas virtuales sobre equipos de escritorio.

7. Telecomandos

[2] En el caso del FS2017 los telecomandos deben ser enviados al segmento de vuelo por el mismo canal en donde se recibe la telemetría, TCP/IP puerto 3210. Los comandos deben ser codificados en una trama AX.25. Para permitir la creación de scripts de comandos, de la misma forma que con la telemetría (6) se utilizaran las capacidades de reflexión para analizar, en tiempo de ejecución los scripts a ejecutar. Los scripts de comandos pueden ejecutarse por acción explicita de un operador o porque fueron aplicados a una pasada. El operador tiene accesos a las bibliotecas de comandos y operación, las cuales están desarrolladas sobre el mismo lenguaje e intérprete de la aplicación. Las bibliotecas tienen pleno

[2] Los módulos que procesan comandos se están en proceso de desarrollo

acceso al ORM de donde puede obtener el diccionario de comandos y los valores de telemetría si necesitara aplicar condicionales que dependieran del estado del segmento de vuelo u otro valor disponible en el sistema.

8. Resultados

El lenguaje (python) es sencillo aprender, pero por sobre todo, está disponible una extensa comunidad con ayudas ante los problemas o desafíos que puedan aparecer. Para las posibles limitaciones es fácil encontrar soluciones alternativas (workarounds). Esto presenta un fuerte contraste con las experiencias realizadas durante la maestría con lenguajes de propósito especifico que requirieron de la asistencia constante de un experto.
Las herramientas de depuración y análisis se mostraron poderosas, incluso superando las expectativas mas allá de las limitaciones en las capacidad de Code Insight/IntelliSense naturales de un lenguaje interpretado.
La solución fue ejecutada en ubuntu linux 15.10 como en windows 10.0 y windows 7.0. Se realizaron pruebas sobre dispositivos físicos y virtuales. En todos casos el software se ejecutó sin cambios. En este punto tanto el interprete como el framework mostraron uno de sus atributos más destacables.
El ORM de Django se mostró estable, se registraron pocos problemas o *bugs* y cuando ocurrieron se encontró información disponible que lo documentaba. Para la creación y modificación de modelos, su flujo divido en dos etapas denominadas *Make Migrations* y *Migrate* permitió el trabajo colaborativo y con múltiples servidores, demostración empírica de la madurez del framework.
El acceso a datos requiere de la práctica de convenciones propias (*Convention over configuration/coding by convention*) que, aunque muy bien documentadas, afectan la curva de aprendizaje. Por otro lado el modelo no acepta atributos privados, cuestión que afecta el encapsulamiento. La capacidad del ORM para trabajar con polimorfismo está limitada y se requiere añadir paquetes especiales, este último punto se pueden considerar como una limitación de relevancia.
Las capacidades de reflexión fueron suficientes para cumplir con los objetivos y los tiempos de respuesta no presentan una limitación a su implementación.La reutilización de métodos previamente cargados colabora en mantener los tiempos entre márgenes aceptables.

9. Conclusiones

Los lenguajes específicos del sector espacial fueron creados hace décadas. Durante los 70, si bien existían opciones interpretadas, están no eran de uso extendido. En la actualidad existen varias opciones (Python, Perl, VBScript, JavaScript) con una amplia cantidad de usuarios de base y múltiples proyectos que avalan su robustez lo que plantea un nuevo escenario a evaluar en lo que respecta a scripts de comandos o calibración de telemetría. Las opciones ampliamente probadas no justifican el desarrollo de un lenguaje propietario, no tener una base de usuarios implica ausencia de documentación, soporte, herramientas

y por sobre todo recursos como explica [11] refiriéndose a ADA.
En términos de persistencia de datos, la vinculación entre el ORM y el motor relacional ha favorecido la productividad. Se ha cuidado de eliminar cualquier consulta directa al motor, para favorecer la compatibilidad con múltiples motores.
Las interfaces aun no se han explotado en su totalidad. Se espera su aplicación efectiva durante las integraciones con diferentes actores o socios del proyecto.
En el estado actual, la solución permite cumplir los objetivos haciendo uso únicamente de componentes COTS y sobre herramientas de propósito general y uso extendido en la industria del software.

Referencias

1. Pablo Soligo, Ezequiel González, Eduardo Sufán, Emmanuel Arias, Ricardo Barbieri, Pablo Estrada, Alfonso Montilla, José Robin, Javier Uranga, M Cecilia Valenti, et al. Misión cubesat fs2017: Desarrollo de software para una misión satelital universitaria. *WICC 2017*, page 843.
2. José J Ramos Pérez. A design for an advanced architecture of satellite ground segments.
3. Ken Galal and Roger P Hogan. Satellite mission operations best practices. 2001.
4. Gonzalo Garcia. Use of python as a satellite operations and testing automation language. In *GSAW2008 Conference, Redondo Beach, California*, 2008.
5. Geraldine Chaudhri, Jim Cater, and Brad Kizzort. A model for a spacecraft operations language. In *SpaceOps 2006 Conference*, page 5708, 2006.
6. TIOBE Software. TIOBE programming community index, september 2017, 2017. [Online; accessed 26-September-2017].
7. L Foti. Corba technology for ground segment system software development. In *DASIA 98-Data Systems in Aerospace*, volume 422, page 303, 1998.
8. Michi Henning. The rise and fall of corba. *Queue*, 4(5):28–34, 2006.
9. Thomas Morel, Gonzalo Garcia, Mike Palsson, and Juan Carlos Gil. High performance telemetry archiving and trending for satellite control centers. In *SpaceOps 2010 Conference Delivering on the Dream Hosted by NASA Marshall Space Flight Center and Organized by AIAA*, page 2111, 2010.
10. Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000.
11. II Smith et al. What about ada? the state of the technology in 2003. 2003.